iDAS SQUIRREL

for

C

System description and user manual

Version 2.1

iDAS® GmbH
Holzheimer Str. 96
65549 Limburg
Germany

Contents

1. Introduction 3

1.1. Purpose 3

1.2. Spellings 4

1.3. Program characteristics 4

2. Installation 5

2.1. SQUIRREL installation on Windows 5

2.2. SQUIRREL installation on Linux 5

SQUIRREL files 5

2.3. Adjustment of the environment for SQUIRREL 6

2.4. Adjustments in the configuration file 6

3. Working with SQUIRREL 7

3.1. SQUIRREL commands 7

3.2. ccinstru: Prepare coverage measurement 7

3.3. Link process and test execution 8

3.4. covprot: Protocol creation 9

3.5. Restrictions with SQUIRREL use 10

4. Test protocol 11

4.1. Setup of the protocol file 11

4.2. Detailed branch logging 13

5. Files created by SQUIRREL 16

6. Example for a SQUIRREL application 17

6.1. Example project 17

6.2. Modification for SQUIRREL 18

6.3. Evaluation of the protocol 19

7. Annex A: Example protocol 20

1. Introduction

1.1. Purpose

While testing a software at the center of attention is the question whether its behaviour is correct. Usually that means checking whether the software responds to test inputs with the expected results. To adjudge the quality of the test process, one must know how representative the amount of test inputs is for the overall function of the software.

Certainly "representative" can not mean that all possible combinations of input data are tried. If two test inputs are so similar that the software internally deals the same way with them, a test of one record suffices. In terms of control flow paths of the software the amount of possible input data resolves thereby into equivalence classes, from which only one record must be tested.

Because of combinational explosion there are a lot of control flow paths by a software, so that a test of all equivalence classes is inexecutable. The branch coverage has proved its worth as a practical approximation of the path coverage: the behaviour of the software is viewed no longer global but local, criterion for the test quality is the rate of program branches reached by test.

SQUIRREL is a tool for evaluation and improvement of the quality of tests for software written in C in the sense discussed above. SQUIRREL determines a code coverage achieved by such a test, that means the rate of program branches and statements which are executed. By the use of logging of branches that have not been passed, SQUIRREL supports the developer in the improvement and completion of the test.

SQUIRREL is mainly intended to be used by system testers, as for component tests more extensive tool support is reasonable (i.e provided by IDAS TESTAT) and for integration tests the code coverage is less important.

The use of SQUIRREL is simple: all of the modules participating in the test are prepared for coverage measurement by replacing a compiler call with a corresponding SQUIRREL command. The modules are linked as usual to executable files, existing tests are started in their test environment. The information gained about the coverage is written to a file and can be edited to be used as a protocol.

1.2. Spellings

< >	:	is a wild card for a name (file name, function name, compiler name, ...). For example, <compiler> is a wild card for the corresponding compiler names.
{ }		optional specification, several specifications are also possible
[ ]	:	optional specification
`Courier`	:	file contents
`Bold`	:	user inputs
italic	:	file names, commands etc,. which are embedded in a stream.

1.3. Program characteristics

Product name:	IDAS SQUIRREL for C
Descriptors:	System test, test of C programs, static analysis, test coverage measurements, white-box protocol, C0 coverage, C1 coverage.
Task:	The product measures the coverage of branches and statements achieved by software tests. All results can be viewed in a protocol.
Responsibilities:	Distribution, information and support: IDAS GmbH Holzheimer Str. 96 65549 Limburg Germany tel: 06431/4040 fax: 06431/40410
Hardware requirements	PC, workstations

2. Installation

The installation of SQUIRREL depends on the operating system used. With Windows and Linux systems the installation is supported by an installation batch/script.

2.1. SQUIRREL installation on Windows

The installation starts by calling an installation batch from the installation directory of the CD via

```
install <destination drive- and directory>
```

The destination directory can be a server directory (also in UNC notation).

While installing SQUIRREL, files are copied to the installation directory, and the license is installed if necessary. SQUIRREL makes no registry entries, it can be simply deinstalled by deleting the installation directory.

2.2. SQUIRREL installation on Linux

The installation starts by calling an installation batch from the installation directory of the CD via

```
sh install.sh <destination directory>
```

SQUIRREL files

SQUIRREL includes the following files:

MS Windows (2000, NT, ME, 98, 95)	**Linux**
covprot.exe ccinstru.exe lizinfo.exe squirrel.cnf xxxcovex.c version2.1 readme.txt	covprot ccinstru lizinfo squirrel.cnf xxxcovex.c version2.1 readme.txt

These files can be installed to any directory of the desired operating system.

In addition, in the subdirectory *beispiel* of the installation directory on the CD there are files of an application example which will be described in this manual later. These files can also be copied to a free selectable directory.

2.3. Adjustment of the environment for SQUIRREL

To work with the tool it is advisable to insert the directory with the installed SQUIRREL files into the search path. It is also possible to call programs *ccinstru* and *covprot* with the full path names. SQUIRREL assumes that the C-Compiler is in the search path.

2.4. Adjustments in the configuration file

The configuration file *squirrel.cnf* provides an opportunity to choose the compiler used. An entry like

```
compiler = <compiler name>
```

should be made in the file for this.

The values permissible for <compiler name> depend on the operating system. The following table shows them in the column "key word":

Operating system	**Key word**	**Compiler**
MS Windows	Bcc32	Borland C++ version 5.x
	Ms-C-110	Microsoft C/C++ version 11.0 and greater (Visual Studio 6.0)
Linux	Gnu	Gnu C compiler *gcc*

3. Working with SQUIRREL

3.1. SQUIRREL commands

Working with SQUIRREL demands two changes of the usual operating cycle:

- preparing the modules for the coverage measurement and
- creation of a protocol

SQUIRREL provides two commands for these steps:

`ccinstru`

and

`covprot.`

The user prepares a coverage measurement for all modules participating in the test by removing the compiler call by a corresponding call of *ccinstru*. The modules are linked to executable files as usual, existing tests are started in their test environment. The information gained about the coverage is written to a file and can be edited later to be used as a protocol.

A detailed description of these steps and of specific applications of SQUIRREL commands will be given in the following chapters.

3.2. ccinstru: Prepare coverage measurement

To prepare a coverage measurement, the SQUIRREL command *ccinstru* has to be called instead of the compiler command for the modules of the tested system which are to be included in the monitoring process. The syntax is:

```
ccinstru [<param>] <module> [-destination directory
=<dir>][-destination=<program>]
```

<module> is the name of a C source file which is part of the tested system; lists of file names are accepted as well. <param> stands for command line parameters of the compiler used (e.g. switch or responce files) according to the compiler manual. <dir> specifies a directory where SQUIRREL stores its temporary files, the default setting is the current directory. To make possible an execution of several programs processed by SQUIRREL at the same time, one can specify with the switch *-destination=<program>* that the name of the file where the coverage information is stored is *<program>.pri.* The default setting is *testcov.pri.*

Attention

The Microsoft- and the gnu-Compiler permit on some switches that the values come straight behind the switch(e.g -I..\dir) or that they can be separated by a space. SQUIRREL supports the first option only.

ccinstru calls the preprocessor for the module, analyses and edits the resulting code and calls the compiler. While calling *ccinstru,* all of the parameters which are not a module name and are not specified as the destination directory are passed on to the preprocessor or compiler. The results of the *ccinstru* call are an object file prepared for the measurement and the SQUIRREL temporary file <*module*>.*brc* which contains information about the branch structure of the module.

While calling *ccinstru,* specifications of response- and configuration files are valid, as far as the compiler permits it. SQUIRREL evaluates these files to produce correct preprocessor and compiler calls.

ccinstru inserts calls of a special function in each branch. This special function is also added to the module. If the module contains a main function, it is complemented with an exit handler (from the file *xxxcovex.c*) which makes sure that the measuring results of all modules are stored at the end of the program.

Modules which are not to be included in the monitoring by SQUIRREL are compiled "regular", without a *ccinstru* call. However, the main module of each program (the module which contains the main function) should be involved in the measurement, because otherwise the measuring results of the other participating modules remain unused.

3.3. Link process and test execution

After the preparation of all relevant modules with *ccinstru*, the object files can be linked to the executable files. There is no need to distinguish between modules compiled with *ccinstru* and software parts which are not in the monitoring(i.e libraries) must not be distinguished.

The link process with SQUIRRELL is normally the same as without it. Only if compiling and link process are combined in a single compiler call, the operation for the use of SQUIRREL must be divided in two steps. On Windows and Unix this is usually made with the compiler switch -c ("compile only") and with the specification of the object files in the linker call.

E.g the combined call

```
bcc32  modul1.c  modul2.c
```

changes in the intermediate step to

```
bcc32 -c  modul1.c   modul2.c
bcc32     modul1.obj modul2.obj
```

and for SQUIRREL use to

```
ccinstru -c  modul1.c   modul2.c
bcc32        modul1.obj modul2.obj
```

After the linking, tests can be performed in their test environment without further modifications. While testing, it is not necessary that the SQUIRREL directory is in the search path, SQUIRREL only needs write permissions in the current directory.

Each program started in the test run writes information about the coverage to a file named *testcov.pri* (*<program>.pri)* in the current directory after its call. The coverage information supplied by a program recall is written to the end of this file, so that the file size increases with the number of program calls. It is advisable to delete the file at the begin of tests, so that results of older and newer test runs (with possibly different sources) do not accumulate.

3.4. covprot: Protocol creation

With the SQUIRREL command *covprot* a protocol based on the data of the files *testcov.pri* (*<program>.pri)* and *<module>.brc* is created.

```
covprot {-tdir=<directory>} [-outlvl=<number>][<program>]
```

With the switch *tdir* the results of test runs can be jointly evaluated in many different directories. *covprot* searches for the temporary files *testcov.pri* in the working directory and in all directories specified with the switch *tdir*, and evaluates them in a common protocol. If the optional parameter for the program name is specified, *<program>.pri* is used instead of *testcov.pri*.

The switch *-outlvl*=<number> specifies how detailed the protocol is. The valid range of <number> is between 0 and 4, the value 0 creates the shortest protocol, the value 4 the most extensive one. The default <number> value is 2.

In detail, the values of the switch *outlvl* result in the following protocols:

outlvl	Output
0	Title page with system overview
1	like 0, in addition a table for all modules
2	like 1, in addition tables of functions for all modules
3	like 2, in addition a table of branches for each function
4	like 3, in addition listing of jumps (goto, return, break, cont) and switches (if, switch) in the branch tables of the functions

The optional parameter <program> should correspond to the value of the switch *-destination* specified with *ccinstru*.

The protocol is written to the file *testcov.prt* (*<program>.prt*) in the current directory.

3.5. Restrictions with SQUIRREL use

SQUIRREL adds its own functions and variables to the processed code. All these symbols begin with the prefix

```
xxxcov_ ...
```

To prevent name conflicts, names in the application tested with SQUIRREL should not start with this prefix.

Another restriction with SQUIRREL use refers to the working with a debugger. After the preprocessor run, SQUIRREL expands the source code with own function calls. However, the debugger only "sees" the code which has not been changed by SQUIRREL. While tracing, this may cause inconsistencies and especially unexpected function calls for which no source code is available .

On the other hand, SQUIRREL leaves the line numbers of the original code unchanged. So, if the debugger is able to work on source code level in spite of the described inconsistencies, one can expect that the displayed source code lines are correct. Therefore it is advisable to execute debugging with "conventionally" built software versions.

4. Test protocol

4.1. Setup of the protocol file

SQUIRREL writes the result of its coverage measurements to the protocol file *testcov.prt* (<*program*>.*prt*). The extent of the protocol can be managed by the value of switch -*outlvl* (See Chapter 3.4).

In the following, the complete protocol created with -*outlvl*=4 is described.

The title page contains the name of the working directory where the protocol was created, time of the protocol creation and the global result of the measurement (branch- and statement coverage) for the complete system. A static analysis of the code monitored by SQUIRREL follows. The analysis supplies the number of functions, statements, branches, goto-statements, the maximum nesting depth and the "cyclomatic complexity"[1].

The second page of the protocol begins with the table of modules sorted alphabetically (with directory). The first column contains the name of the module (without directory). In the next columns the values for the number of statements, branches and complexity follow. These three numbers are summed up for all functions of the module. The fifth column of the table contains the maximum nesting depth in the functions of the module. The two last columns display the measured coverage in terms of statements and branches. This table can only be viewed with *outlvl* greater than 0.

If *outlvl* is greater than 1, the following pages of the protocol display a hierarchical system of tables. On the highest level each module starts a new table on a new page. On the next level the first table contains an overview of the functions of the module. It shows for each function the same information as in the module table. The next tables describe the passes through the branches for each function.

These function tables are only displayed if the *outlvl* is greater than 2.

The values listed in the protocol are specified as follows:

1) Function
   A function in the source code is defined by the construct <function definition> according to the Ansi-C standard.
2) Statement
   A statement in the source code is defined by the construct <statement> according to the Ansi-C standard.

[1] McCabe, A Complexity Measure, Transactions on Software Engineering, Vol. SE-2, No 4, Dec 1976, p 308

Exception: SQUIRREL does not include the special construct <statement><compound statement> to the statements.

3) Branch

According to our definition, branches are produced both on decisions (primitive branches) and after merging of branches separated before. In addition, each function start is considered to be a start of a branch. Parsing through all primitive branches implicates running through all branches if they can be reached dynamically.

Example: At the beginning of an "if" construct, two new branches are created (indicated by "then" and "if") which are merged after the "if" construct to a branch again.

All branches are listed in the next chapter.

4) Goto statement

A goto statement is defined by the construct <jump-statement> according to the Ansi-C standard.

5) Nesting depth

Each function branch begins with depth 0, an "if"-, "switch"-, "conditional"-, "for"-, "while"- or "do...while"- construct increments the depth by 1. After the constructs the nesting depth is reduced again. "if", "switch", "conditional" have in the protocol the same nesting depth as "then"-, "else"-, "case"-, "default"-, "conditionalthen"- and conditionalelse" branches.

6) Complexity

Each function gets the value 1 first. An "if", "for", "while", "do...while" and "?:" increments the value of the complexity by 1.

"switch" statements increment the value by the number of their "case" branches (without the default branch). "case" labels which are on the same statement are merged to one case branch.

Attention

Modules prepared for a measurement with *ccinstru* are **not** displayed in the protocol if they were not used by any program (no function defined in the module was called).

4.2. Detailed branch logging

The tables which contain the result of the coverage measurement for the branches of individual functions make up the biggest part of the complete protocol. A table is created for each function, its lines represent control flow branches, its columns supply static and coverage information. Each function branch has the following entries:

Column title	**Meaning of the column**
Line	Line number in the module where the function or the branch begins.
Lv	The nesting depth of the branch.
Type	Specifies the branch type. A list of possible branch types follows below.
Passes	Specifies the number of passes through the branch. Branches which have not been passed are highlighted with oooooo or ..... . The second designation means that the branch was not passed because the superordinate branch had been not passed.
First pass	Date and time of the first pass through this branch. Format: <day><month><hour>:<minute>
Last pass	Date and time of the last pass through this branch. Format: <day><month><hour>:<minute>

SQUIRREL distinguishes the following branch types (field *type* of the protocol):

Abbrevation	**Branch type**
Primitive branches	
function	Function branch
then else quthen quelse	True and false branch of an if or "?:" operator.
case default	Case and default branch of a switch statement. On the switch branches the number is specified (beginning with 0).
ifnot nonalt	A false branch of an if statement or a default branch of a switch statement which has not been programmed (added by SQUIRREL).
while for contdo	Loop pass. These branches are passed if the control statement of the according loop construct of C decides positive on the continuation of the loop.

endwhile
endfor
enddo
	Loop end. These branches are passed if the loop is left regular (that means through the loop control statement but not through jumps). They contain no statements.

Branches which are created by merging.	
res <branch>	Resumption of the last branch of the same nesting depth after an if-, switch- or loop construct. The branch must be counted separatly, because it can be skipped by jump statements in the nested construct. <branch> is the abbreviation of the branch which is resumed here.
do	Branch beginning at the start of a do...while loop. This branch is always passed if the do...while loop is reached and must be distinguished from the "contdo" branch which is passed only if the do...while loop is continued.
label	Branch beginning by a label.

Abbrevation	Branch type
Additional orientation guide in the white box protocol	
if switch qumark	Switches qumak represents the"?:" operator of C.
return goto break cont	Unconditional jump statements. cont represents the "continue" statement of C.

If conspicious combinations are found in the white box protocol, warnings are produced. They appear on the left of the first column in the white box protocol. Three warnings can occur:

Abbreviation	Meaning
C	Case check warning A default branch which has not been programmed was passed.
S	Sum check warning The numbers of passes through a subordinate and the superordinate branch do not match. This is generally an effect of a jump statement (goto, return, break, cont).
L	Loop check warning A loop was passed just as often as the superordinate branch - the loop is possibly dispensable.

5. Files created by SQUIRREL

The following table shows all files which are created or changed by SQUIRREL:

File name	Written by	Read by	Content
<module>.brc	ccinstru	covprot	Information about the branch structure of the module.
<module>.ins <module>.zwg	ccinstru	ccinstru	Temporary files which are deleted at the end of a faultless pass.
<module>xxxcov_t.c	ccinstru	ccinstru	Temporary files which are deleted at the end of a faultless pass.
<module>.cfg <module>.rsp xxxcov.cfg xxxcpp.cfg xxxcov.rsp	ccinstru	preprozessor, compiler	Generated configuration- and responce files (specific to the compiler). They are deleted at the end of a *ccinstru* pass.
<module>.i	preprozessor, ccinstru	compiler	Source code after the preprocessor and trace-preparator pass. It is deleted at the end of *ccinstru* pass.
testcov.pri or <program>.pri	Instrumented user software	covprot	Coverage information, internal format
testcov.cov or <program>.cov	covprot	covprot	Temporary file which is deleted at the end of a faultless pass.
testcov.prt or <program>.prt	covprot		Protocol output

6. Example for a SQUIRREL application

6.1. Example project

As example for the working of SQUIRREL, the small program *xc* is presented now. The program displays numbers in different formats (binary, decimal, hexadecimal, octal, ASCII) which are entered in a dialog or as command line parameters. The input can be made in the same formats as the output (except of binary), *xc* handles them automatically correct.

The project consists of three source files (*xc.c, bin.c, bin.h*), a batch file for the test (*xctest.bat* for Windows or *xctest.sh* for Unix) and two makefiles (*xc.mak* without SQUIRREL call and *xcs.mak* with it). These files are located in the subdirectory *beispiel* of the installation directory of the SQUIRREL CD according to the operating system.

Borland C++ 5.x is specified as the default compiler for Windows, for Unix it is the Gnu compiler. To use another compiler, the configuration file *squirrel.cnf* and the compiler- and linker calls in the makefiles have to be changed. The explanation is made on the Windows example. But the steps on Unix are identical. All commands (indented lines of the makefiles) can also be specified manually.

xc.mak is a project makefile without SQUIRREL use:

```
#  Makefile for the example project xc
#  build and system test under Windows

# perform system test
xctest.log : xc.exe xctest.bat
  xctest.bat

# link xc
xc.exe : xc.obj bin.obj
  bcc32 xc.obj bin.obj

# compile xc.c
xc.obj : xc.c bin.h
  bcc32 -c xc.c

# compile bin.c
bin.obj : bin.c bin.h
  bcc32 -c bin.c
```

This makefile controls both compiling and link process of the program and a small system test. *xctest.bat* calls the program with different parameter combinations and directs the output to *xctest.log*. An automatic evaluation of this test, e.g by comparing *xctest.log* with target data, is not contained in this example.

6.2. Modification for SQUIRREL

To evaluate the quality of the system tests in *xctest.bat* and to receive information about necessary completions, the test is complemented by a SQUIRREL logging now. The following changes are necessary for this:

- The compiler calls (of the two last blocks in *xc.mak*) are replaced by the calls of *ccinstru*.
- The linker call (the second block) remaines unchanged. (The linker call being performed with the same command as the compiler call does not change anything).
- A SQUIRREL protocol is created with *covprot* after a performed test.

The outcome is the modified makefile *xcs.mak*:

```
#  Makefile for the example project xc
#  build and system test with Squirrel on Windows

# perform system test and create Squirrel protocol
xctest.log : xc.exe xctest.bat
  xctest.bat
  covprot -outlvl=4

# link xc
xc.exe : xc.obj bin.obj
  bcc32 xc.obj bin.obj

# modify and compile xc.c
xc.obj : xc.c bin.h
  ccinstru -c xc.c

# modify and compile bin.c
bin.obj : bin.c bin.h
  ccinstru -c bin.c
```

Both on Windows and Linux the compiling and test run can be started with the command:

```
make -f xcs.mak
```

This call should create the SQUIRREL protocol of the test of *xc.exe* (If you have already tried the compiling and test without SQUIRREL, you should delete the object files and start the make commands again).

6.3. Evaluation of the protocol

After running the makefile, the SQUIRREL protocol *testcov.prt (<program>.prt)* is available in addition to executable files, object files and the log file of the system test. If the switch *-outlvl* is set to 4 while calling *covprot*, the file contains a detailed white box protocol of all the functions of the software in addition to a title page and an overview of the module coverage.

This protocol is printed in the annex.

The title page provides an overview of the overall result and the static properties of the software. All numbers refer to the entire software modified, 2 modules with 3 functions in this example. 98% of all statements of the software have been executed and 96% of its control flow branches have been passed.

The second page contains an itemisation of the total numbers of the first page according to modules. One can see that in the module *xc* all statements and branches have been passed. In the module *bin* only 90% of the statements have been executed, the branch coverage is still lower.

To identify branches which have not been covered, the following pages should be viewed. These pages contain a coverage itemisation ordered by modules at the function- and branch level. As example the module *bin* (with the function having the same name) is viewed on page 3 of the protocol.

The first table shows the coverage statistics for the functions of the module. As the module contains only one function, the data for the module and the function are identical. The second table shows the coverage at a branch level. The string "oooooo" in the column *passes* demonstrates that the last "if" of the function has never been true.

After inspection of the missing branch in the source code, a missing test case can be constructed. The task of the function *bin* is the binary representation of integers with null suppression. The identified branch handles the case that the input is zero. Then the software displays one digit ‘0’. So, the missing test case is a call of *xc* with the argument 0.

7. Annex A: Example protocol

```
IDAS-SQUIRREL                                          Coverage measurement
------------------------------------------------------------------------------


                    ***** TEST COVERAGE *****


                   Directory : i:\testat\squirrel\nt


                Coverage measurement at 31-Aug-2001 11:19


                       System test coverage

                Coverage of statements: C0 =  98 %
                Coverage of branches  : C1 =  96 %
                      Static analysis of the system

          Number of functions                              :     3
          Number of statements                             :    40
          Number of branches                               :    45
          Number of jumps                                  :     0
          Max. nesting depth                               :     3
          Average quantity of complexity (McCabe)/ function :   6,0









------------------------------------------------------------------------------
testcov.prt                                                  Page   1+
```

```
IDAS-SQUIRREL                                                Coverage measurement
------------------------------------------------------------------------------


start modul_list ( total coverage : C0 = 98 % C1 = 96 % )

                                   |   total    | Mc   | max |
 module                            | stmt brch  | Cab  | lvl |  C0   C1
-----------------------------------|------------|------|-----|--------
 bin                               |   10    9  |   4  |   2 |  90   89
 xc                                |   30   36  |  14  |   3 | 100   97

end modul_list
```

```
IDAS-SQUIRREL                                              Coverage measurement
------------------------------------------------------------------------------


start modul bin ( C0 = 90 % C1 = 89 % )

                                   |   total    | Mc  | max |
 function                          | stmt brch | Cab | lvl |  C0  C1
-----------------------------------|-----------|-----|-----|--------
 bin                               |   10    9 |   4 |   2 |  90  89



start function bin ( C0 = 90 % C1 = 89 % )
                        |          |    first      |    last
  line lv type          |  passes  |    pass       |    pass
------------------------|----------|---------------|-------------
    13    function      |       16 | 31-Aug 11:19 | 31-Aug 11:19
    19  1 for           |      256 | 31-Aug 11:19 | 31-Aug 11:19
    22  2 if            |
    22  2 then          |      108 | 31-Aug 11:19 | 31-Aug 11:19
    23  2 else          |      148 | 31-Aug 11:19 | 31-Aug 11:19
    19  1 endfor        |       16 | 31-Aug 11:19 | 31-Aug 11:19
    26  0 res function  |       16 | 31-Aug 11:19 | 31-Aug 11:19
    26  1 if            |
    26  1 then          |   oooooo |
        1 ifnot         |       16 | 31-Aug 11:19 | 31-Aug 11:19
    27  0 res function  |       16 | 31-Aug 11:19 | 31-Aug 11:19
    28  0 return        | RETURN
end function bin

end modul bin
```

```
IDAS-SQUIRREL                                              Coverage measurement
------------------------------------------------------------------------------


start modul xc ( C0 = 100 % C1 = 97 % )

                                    |   total    | Mc  | max |
 function                           | stmt brch | Cab | lvl |  C0  C1
------------------------------------|-----------|-----|-----|--------
 parmread                           |   15   19 |   7 |   1 | 100 100
 main                               |   15   17 |   7 |   3 | 100  94


start function parmread ( C0 = 100 % C1 = 100 % )
                        |           |    first     |    last
  line lv type          |  passes   |    pass      |    pass
------------------------|-----------|--------------|-------------
    20    function      |       20  | 31-Aug 11:19 | 31-Aug 11:19
    27  1 if            |
    27  1 then          |        4  | 31-Aug 11:19 | 31-Aug 11:19
    27  1 return        | RETURN
        1 ifnot         |       16  | 31-Aug 11:19 | 31-Aug 11:19
S   28  0 res function  |       16  | 31-Aug 11:19 | 31-Aug 11:19
    28  1 for           |       13  | 31-Aug 11:19 | 31-Aug 11:19
    28  1 endfor        |       16  | 31-Aug 11:19 | 31-Aug 11:19
    29  0 res function  |       16  | 31-Aug 11:19 | 31-Aug 11:19
    29  1 for           |       47  | 31-Aug 11:19 | 31-Aug 11:19
    29  1 endfor        |       16  | 31-Aug 11:19 | 31-Aug 11:19
    30  0 res function  |       16  | 31-Aug 11:19 | 31-Aug 11:19
    30  1 if            |
    30  1 then          |       10  | 31-Aug 11:19 | 31-Aug 11:19
        1 ifnot         |        6  | 31-Aug 11:19 | 31-Aug 11:19
    31  0 res function  |       16  | 31-Aug 11:19 | 31-Aug 11:19
    31  1 if            |
    31  1 then          |       12  | 31-Aug 11:19 | 31-Aug 11:19
    31  1 return        | RETURN
        1 ifnot         |        4  | 31-Aug 11:19 | 31-Aug 11:19
S   32  0 res function  |        4  | 31-Aug 11:19 | 31-Aug 11:19
    33  1 if            |
    33  1 then          |        2  | 31-Aug 11:19 | 31-Aug 11:19
    33  1 return        | RETURN
        1 ifnot         |        2  | 31-Aug 11:19 | 31-Aug 11:19
S   34  0 res function  |        2  | 31-Aug 11:19 | 31-Aug 11:19
    35  0 return        | RETURN
end function parmread




------------------------------------------------------------------------------
testcov.prt                                                        Page    4+
```

```
start function main ( C0 = 100 % C1 = 94 % )
                      |          |    first     |    last
  line lv type        |  passes  |    pass      |    pass
----------------------|----------|--------------|-------------
    44    function    |        2 | 31-Aug 11:19 | 31-Aug 11:19
    52  1 while       |       16 | 31-Aug 11:19 | 31-Aug 11:19
    54  2 qumark      |
    54  2 quthen      |   oooooo |
    54  2 quelse      |       16 | 31-Aug 11:19 | 31-Aug 11:19
    54  2 if          |
    54  2 then        |        8 | 31-Aug 11:19 | 31-Aug 11:19
    55  2 else        |        8 | 31-Aug 11:19 | 31-Aug 11:19
    55  3 for         |       16 | 31-Aug 11:19 | 31-Aug 11:19
    55  3 endfor      |        8 | 31-Aug 11:19 | 31-Aug 11:19
    56  1 res while   |       16 | 31-Aug 11:19 | 31-Aug 11:19
    57  2 for         |       32 | 31-Aug 11:19 | 31-Aug 11:19
    57  2 endfor      |       16 | 31-Aug 11:19 | 31-Aug 11:19
    58  1 res while   |       16 | 31-Aug 11:19 | 31-Aug 11:19
    59  2 for         |       48 | 31-Aug 11:19 | 31-Aug 11:19
    59  2 endfor      |       16 | 31-Aug 11:19 | 31-Aug 11:19
    60  1 res while   |       16 | 31-Aug 11:19 | 31-Aug 11:19
    52  1 endwhile    |        2 | 31-Aug 11:19 | 31-Aug 11:19
    62  0 res function|        2 | 31-Aug 11:19 | 31-Aug 11:19
    62  0 return      | RETURN
end function main

end modul xc
```